Name: Andrew Hayes
Student ID: 21321503
Programme: 4BCT

CT4101 2024–11–03

Assignment 2: Regression Using Scikit-Learn

1 Description of Algorithms

1.1 Algorithm 1: Random Forest

Random decision forest is a supervised machine learning algorithm that can be used for both classification & regression that builds upon the **decision tree** algorithm by combining several decision trees to generate predictions for a dataset. An implementation of this algorithm for regression is provided in scikit-learn as `sklearn.ensemble.RandomForestRegressor`[4].

Since the random decision forest algorithm builds upon the decision tree algorithm, it is first necessary to explain briefly what decision trees are and how they work. A decision tree can be thought of as a series of internal nodes (i.e., nodes which are not leaf nodes) that contain a question which separates the input data. The decision tree is traversed from root to leaf for each instance being predicted, where the leaf node to which we arrive provides the predicted value for that instance. For example, a decision tree might be used to predict the price of a house, where each internal node is a question that helps to separate houses of different values, and each leaf node provides a predicted price. Each internal node should narrow down the final prediction as much as possible, i.e., each question should provide the maximum information about the instance and should be arranged in the order that narrows it down as quickly as possible.

Decision trees have many advantages: they are visualisable by humans and aren't "black-box", they can model non-linear relationships easily, and they are robust to outliers. However, they have their disadvantages, including instability (small changes in the training data can significantly alter the tree structure) and in particular **overfitting**: when the algorithm fits too exactly to the training data, making it incapable of generalising to unseen data.

Random forests work by combining many decision trees into a forest, thus improving accuracy & reducing overfitting by averaging multiple trees, reducing variance, and leading to better predictions. These decision trees are each generated using random, potentially overlapping subsets of the training data. While the original random forest algorithm worked by taking the average of the predictions decided on by the set of trees[1], the scikit-learn `RandomForestRegressor` works by taking the mean of the predictions from each tree to arrive at the final output value[2].

In `RandomForestRegressor`, each tree is generated as follows:

1. A subset of the training data is randomly selected (hence the "Random" in the name of the algorithm). These subsets are selected "with replacement" which means that different trees can select the same samples: they are not removed from the pool once they are first selected. This results in unique, overlapping trees.
2. Starting with the root node, each node is *split* to partition the data. Instead of considering all features of the samples when choosing the split, a random subset of features is selected, promoting diversity across the trees. The optimal split is calculated using some metric such as mean squared error to determine which split will provide the largest reduction in prediction error.
3. This process is repeated at every node until no further splits can be made.

I chose the random forest regressor because it is resistant to overfitting, works well with complex & non-linear data like the dataset in question, handles both categorical & numerical features, and offers a wide variety of hyperparameters for tuning. It also has many benefits that are not particularly relevant to this assignment but are interesting nonetheless: it can handle both regression & classification tasks, can handle missing data, and can be parallelized for use with large datasets.

1.1.1 Hyperparameter 1: `n_estimators`

The hyperparameter `n_estimators` is an ***`int`*** with a default value of 100 which controls the number of decision trees (*estimators*) in the forest[4]. Increasing the number of trees in the forest generally improves the model's accuracy & stability, with diminishing marginal returns past a certain value, at the trade-off of increased computation & memory consumption. Each tree is independently trained, so there is a significant trade-off between computational cost & performance. Using a lower number of estimators can result in underfitting, as there may not be enough trees in the forest to capture the complexity of the data.

1.1.2 Hyperparameter 2: `max_depth`

The hyperparameter `max_depth` is an ***`int`*** with a default value of **`None`** which controls the maximum "depth" of each of the trees in the forest[4], where the "depth" of a tree refers to the longest path from the root node to a leaf node in said tree. With the default value of **`None`**, the trees will continue to grow until they cannot be split any further, meaning that each leaf node either only contains samples of similar values or contains a number of samples lower than the `min_samples_split` hyperparameter. The `min_samples_split` hyperparameter determines the minimum amount of samples needed for a node to be split; it has a default ***int*** value of 2 and therefore, since I am not tuning this hyperparameter for this assignment, it has no relevance as any leaf node that doesn't reach the minimum amount of samples to be split is a "pure" node by virtue of containing only one value.

High `max_depth` values allow for the trees to capture more complex patterns in the data, but can overfit the data, leading to poor prediction accuracy. Bigger trees also naturally incur higher computational costs, requiring both more computation to create and more memory to store. Lower `max_depth` values result in simpler trees which can only focus on the most important features & patterns in the data, which in turn can reduce overfitting; however, low values run the risk of creating trees which are not big enough to capture the complexity of the data, and can lead to underfitting.

1.2 Algorithm 2: Gradient Boosting

Gradient boosting is a supervised machine learning algorithm that can be used for both classification & regression tasks. It builds upon the **decision tree** algorithm by combining multiple decision trees sequentially, with each new tree trained to correct the errors of the previous ones. This sequential training process is distinct from that of **random forests**, which aggregates many trees independently trained on random subsets of the data. An implementation of this algorithm for regression is provided in scikit-learn as `sklearn.ensemble.GradientBoostingRegressor`[3]. While gradient boosting can also be used for classification, I will only refer to its use as a regression algorithm in this assignment, as classification is not relevant to the specific task at hand.

While both random forest and gradient boosting use multiple decision trees, they differ fundamentally in how they build and combine these trees:

- In **Random Forest**, trees are built independently on random subsets of data and features, which helps reduce variance and improves generalisation. The final prediction is obtained by averaging the outputs of each tree (for regression tasks) or by majority voting (for classification tasks).
- In **Gradient Boosting**, trees are built sequentially, with each new tree trained to correct the errors of the previous ones. This iterative process focuses on reducing bias, as each tree addresses the residual errors from prior iterations. The final prediction is obtained by summing the predictions of each tree, where later trees contribute to refining the overall model.

In `GradientBoostingRegressor`, the trees are generated as follows:

1. The algorithm initializes the model with a constant value, which serves as the baseline prediction for all instances.
2. At each iteration, the algorithm computes the negative gradient of the loss function with respect to the current model's predictions. This gradient indicates the direction in which to adjust the predictions to minimize the loss.
3. A decision tree is then fitted to the negative gradients (i.e., the errors) from the previous iteration, where each tree is constrained in complexity (often by limiting the depth) to avoid overfitting.
4. The predictions from this new tree are scaled by a learning rate and added to the ensemble's predictions to update the model.

I chose the gradient boosting regressor because it is effective for complex and non-linear datasets, provides robust performance on diverse types of data, and offers flexibility with numerous hyperparameters for tuning. It is also an interesting comparison to the random forest regressor, as they both are ensemble methods that use multiple decision trees.

1.2.1 Hyperparameter 1: `n_estimators`

The hyperparameter `n_estimators` is an ***int*** with a default value of 100, which controls the number of boosting stages (i.e., decision trees) added to the model[3]. Increasing the number of estimators generally improves model accuracy and stability, but it may lead to overfitting if too many trees are added. Thus, there is a trade-off between performance and computational cost, as more estimators require more training time and resources.

1.2.2 Hyperparameter 2: `max_depth`

The hyperparameter `max_depth` is an ***int*** with a default value of 3, controlling the maximum depth of each tree in the ensemble[3]. By limiting the depth, the algorithm helps prevent overfitting, as shallower trees capture more general patterns in the data.
High `max_depth` values allow trees to capture complex patterns, but can lead to overfitting and reduced generalisation on unseen data. In contrast, lower `max_depth` values create simpler trees that focus on the most important features and relationships, which can enhance generalisation but may result in underfitting if the model fails to capture necessary data complexity.

2 Model Training & Evaluation

2.1 Algorithm 2: Gradient Boosting

2.2 Algorithm 2: Gradient Boosting

3 Conclusion

References

[1] Leo Breiman. “Random Forests”. In: *Machine Learning* 45 (2001).

[2] scikit-learn Documentation. *Ensembles: Gradient boosting, random forests, bagging, voting, stacking*. URL: `https://scikit-learn.org/stable/modules/ensemble.html` Accessed on: 2024-11-03.

[3] scikit-learn Documentation. *`GradientBoostingRegressor` API Reference*. URL: `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html` Accessed on: 2024-11-03.

[4] scikit-learn Documentation. *`RandomForestRegressor` API Reference*. URL: `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html` Accessed on: 2024-11-03.